

- xchg 成功 → 立即进入临界区，开锁很小
- 缓慢的 slow path
- xchg 失败 → 浪费 CPU 自旋等待

互斥锁 (通过系统调用访问 locked)

- 更经济的 slow path
 - 上锁失败线程不再占用 CPU
- 更慢的 fast path
 - 即锁上锁成功也需要进出内核 (syscall)

Futex: Fast Userspace muTexes

小孩子才做选择，操作系统里是全都要用！

- Fast path: 一条原子指令，上锁成功立即返回
- Slow path: 上锁失败，执行系统调用阻塞
- 性能优化的最佳实践
 - 看 average (frequent) case 而不是 worst case

POSIX 线程库中的互斥锁 (pthread_mutex)

- 观察线程库中的 lock/unlock 行为
- 1. Mutex 没有争抢的情况
- 2. Mutex 有争抢的情况

- 容忍机器总线 (Partition tolerance)
 - 单机程序目标，尽可能多的服务并行的请求
- QPS: 吞吐量
- Tail latency: 一个请求慢了，其他的请求不能慢。

因此，线程不够用了，他能并行但是很慢，远多于处理器的线程会导致性能问题。而且切换开销，维护开销很大，所以引出了协程。传统的语言机制也许不是很适合开发，所以出现了go。

- AI模型 (计算密集 + 数据密集)
- 浏览器

有模型，就可以模拟。很多任务，具有空间局部性。天然就是并行的，可以不断地分成小部分。物理世界，交互出现在局部性。

GO 和 GOROUTINE

Go: 小孩子才做选择，多线程并行和轻量级开发我全都要！

Goroutine: 概念上类似线程，实际是线程和协程的混合体

- 每个 CPU 上有一个 Go Worker，自由调度 goroutines
- 执行到 blocking API 时 (例如 sleep, read)
 - Go Worker 继续改成 non-blocking 的版本
 - 成功 → 立即继续执行
 - 失败 → 立即 yield 到另一个需要 CPU 的 goroutine
 - 才 25 秒了！CPU 和操作系统全部用到 100%

Go 语言中的同步

Do not communicate by sharing memory; instead, share memory by communicating. — Effective Go

共享内存 = 万恶之源

- 信号量/事件变量，实现了同步，但没有实现“通信”
 - 数据传递完全靠手工 (使上锁就错了)

但 UNIX 时代就有一个实现并行的机制了

- `cat > /dev /dev /dev`
 - 管道是一个天然的生产者/消费者！
 - 为什么不不用“管道”实现协程/线程间的同步 + 通信呢？
 - Channels in Go

传统代码都是共享内存，但是实际上不能实现传递信号的作用。

如何解决死锁？（事实上，面对死锁，我们在代码层面基本只能使用上锁顺序来解决。前三个假设都是破坏掉了互斥的假设）

- Mutual-exclusion
- Wait-for
- No-preemption (非抢占)
- Circular-chain

数据竞争 (data race)since C11 Undefined behavior) 【不同的线程同时访问同一内存，且至少有一个是写】

- Weak memory model

在实际中，最常見出現的就是，

1. 上错了锁
2. 忘记上锁。

“内存”可以是地址空间中的任何内存

- 可以是全部变量
- 可以是堆区分配的变量
- 可以是栈

“访问”可以是任何代码

- 可能发生在你的代码里
- 可以发生在框架代码里
- 可能是一行你没有读到过的汇编代码
- 可能时一条 ret 指令

97%的bug

- 原子性被破坏 (ABA) 【互斥锁】
- 顺序被破坏 (BA) 【条件变量/信号量】

人类是sequential creature，并发机制后果自负。